THE SERIAL LOG-MACHINE

by

Pey-yun Peggy Li

Technical Report #4517

Department of Computer Science

California Institute of Technology

May 30, 1981

I. INTRODUCTION

The one-chip computer is no longer a dream due to the rapid development of VLSI technology. More and more transistors can be put on one chip, and in the near future, we can expect to see more and more processors on one chip. Microprocessors evolved from 4-bit processors to 8-bit and 16-bit processors. The improvement of the performance makes microprocessors more and more attractive. Building a microprocessor that includes a floating-point processor will soon be a reality. Floating-point numbers are widely used in any kind of scientific application. It is not realistic to ignore this part when building a real, useful computer. Since the hardware cost of a floating-point processor is too high, floating point operations are usually provided by a firmware-based (microprogramming) approach or a software-based approach in microcomputer/minicomputer systems. In large computers or some versions of minicomputers, an additional floating-point processor, composed of a few floating-point registers and two different arithmetic units which handle the operations of the exponent part and the mantissa part respectively, is connected and operated in parallel with the ordinary ALU.

Because of the relative complexity of multiplication and division in a floating-point number system, I have investigated the logarithmic number system as a possible replacement of the conventional floating-point representation. I devised an algorithm to do addition and subtraction of numbers in logarithmic representation. I also designed a chip to perform the four basic arithmetic operations on numbers represented by their logarithms. My chip, which I named as "the serial logarithm machine", is a small serial processor including two one-bit ALU's, five 64-bit shift registers for each ALU, two tables which contain the value of $\log_2(1+2^{**}(-m))$ and $\log_2(1-2^{**}(-m))$, where m is an integer and $0<m<27$, and a six-bit counter and a PLA controller for each ALU. For the time being, only the data path and the controller part are included in the chip. The instruction set includes four basic arithmetic operations, logic and unary operations, including shift, negate and increment/decrement, etc. It is sufficient to implement my algorithm but not for a complete computing machine.

In the next section, I will describe the logarithmic representation and its four arithmetic operations. The invariant method, the algorithm for the addition/subtraction of logrithm numbers, will be explained in detail. The floorplan and each basic cell of the chip will be illustrated in Section III and the instruction set I constructed and the microcode of the PLA controller will be presented in Section IV. The timing scheme of

this chip is illustrated in Section V and a simulation result for this machine is appended at the last section.

II. LOGRITHMIC REPRESENTATION AND ITS ARITHMETIC OPERATIONS

2,1 LOGRITHM REPRESENTATION

The logarithm number system was introduced to the computer world by Kingsburg and Rayner in 1971[1]. The algorithms for the logarithm arithmetic, especially for the logarithm addition were described in that paper. After that, similar work was carried out by Swartzlander and Alexopoulos in 1975[2] and Lee and Edgar in 1977 [3].

Logarithm representation is a good method to represent a wide range of numbers. Instead of dividing a number into two parts (mantissa and exponent) in the floating point representation, the logarithm number system just treats a number, no matter which is very large or very small, as a fixed point number.

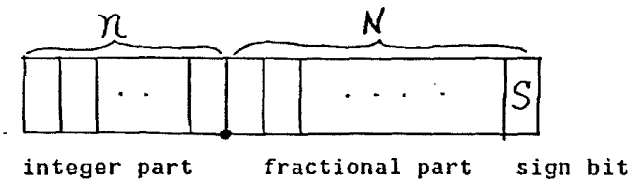
The logarithmic representation of a given number A contains a sign bit S_A ,where

$$S_A = \begin{cases} 1 & \text{for } A < 0 \\ 0 & \text{for } A \geq 0 \end{cases}$$

To make sure that the value after taking the logarithm will be positive, the excess-K method is used. The second part of the representation is

$$L_A = \begin{cases} K+\log_p|A| = \log_p(p^K * |A|) & \text{for } |A| > 1/p^K \\ 0 & \text{for } |A| \leq 1/p^K \end{cases}$$

Assume the form of the logarithm representation is a fixed-point number with a sign bit. The part at the left side of the decimal point is called integer part and the part right to the decimal point is called fractional part. The sign bit is put at the right of the LSB. If the length of the integer part is n, the length of the fractional part is N, then the excess amount K is equal to 2**(n-1).



So the range of the number which can be represented by this number system is

$$|A| < p^{**}(2^{n-1})$$

For example, zero or any number whose absolute value is less than $p^{**}(-2^{n-1})$
is represented as 00....0.00....00--0
1 is represented as 10....0.00....00--0
and the maximal number is 11....1.11....11--0 ← sign bit

The finite length of the fractional part introduces a round-off error which is a constant relative error. The maximum error of such a representation is (1/2)LSB, i.e. (1/2)*(2**(-N))=2**(-N-1). If $\log_p A$ is the accurate value without any truncation, then the value of the finite length number is within the range of $\log_p A+2^{**}(-N-1)$ (let's ignore the excess value K). Take the exponential of this number, then we get the value which the finite-length logarithm number represents:

$$p^{(\log_p A \pm 2^{**}(-(N+1)))} = A^{*}p^{\pm 2^{**}(-(N+1))}$$

By subtracting the accurate value A from this equation, the rounding error is $A^{*}(p^{\pm 2^{**}(-(N-1))}-1)$. Obviously, this is a constant relative error no matter what value A is. The accuracy depends on N, the length of the fractional part.

2.2. MULTIPLICATION/DIVISION IN LOGARITHMIC REPRESENTATION

Multiplication and division of two numbers are just addition and subtraction in the logarithmic representation.

Given $A'=K+\log_p(A)$
$B'=K+\log_p(B)$

where K is the excess constant
then

$$C'=K+\log_p(A*B)$$
$$=(K+\log_p(A))+(K+\log_p(B))-K$$
$$=A'+B'-K$$

and

$$D'=K+\log_p(A/B)$$
$$=(K+\log_p(A))-(K+\log_p(B))+K$$
$$=A'-B'+K$$

No error will be generated except the rounding error which is the sum of errors of these two numbers. The maximum is $\pm 2^{**(-N)}$ corresponding to (A+B) or (A-B), the relative maximum error will be $(p^{2^{**(-N)}}-1)$.

For a serial arithmetic unit, add/subtract two numbers bit by bit takes M bit time by using an adder/subtractor, where M is the word length of that representation (M=N+n). The sign bit is located at the rightmost position and we should test two sign bits first to check whether they are in the same sign or not. It takes only 1 bit time by using a exclusive-OR gate. At last, we should add/subtract K to/from the result. Since K only changes the MSB of that word, we can handle that problem as well as overflow or underflow by adding a small piece of machinery and it takes one more bit time. So the total time for multiplication or division is (M+2)-bit time.

2.3. ADDITION/SUBTRACTION IN LOGARITHMIC REPRESENTATION

The slow and the complexity of addition and subtraction are the major weaknesses of logarithmic representation. This is also the reason why the logarithmic number system can't replace the conventional binary number system in the computer world, even though the former has several good characteristics which are much better than the floating-point representation.

In the following, the algorithms of addition and subtraction are given and compared for the trade-off of time and space.

Given

$$A'=K+\log_p(A)$$
$$B'=K+\log_p(B)$$

then

$$C'=K+\log_p(A+B)$$

$$
\begin{aligned}
&= K+\log_p A*(1+(B/A)) \\
&= K+\log_p A+\log_p(1+(B/A)) \\
&= K+\log_p A+\log_p(1+p^{**}(\log_p B-\log_p A)) \\
&= K+\log_p A+\log_p(1+p^{**}(B'-A')) \\
&= A'+\log_p(1+p^{**}(B'-A')) \qquad \text{for } A \geq B
\end{aligned}
$$

or

$$
\begin{aligned}
C' &= K+\log_p(A+B) \\
&= K+\log_p B*(1+(A/B)) \\
&= K+\log_p B+\log_p(1+p^{**}(\log_p A-\log_p B)) \\
&= B'+\log_p(1+p^{**}(A'-B')) \qquad \text{for } A < B
\end{aligned}
$$

Assume $x=|A'-B'|$, then the only thing we need to compute is

$$\log_p(1+p^{-x}) \quad \text{where } x \geq 0 \qquad (1)$$

Invariant Method

The conventional way to solve equation (1) is table look-up. The table length will grow exponentially with the word length of the logrithmic number. Hence, it is impractical for a long word length representation such as 64 bit which I chose for my chip.

The method I chose to solve equation (1) is to compute it directly, i.e. calculate p^{-x} first, add or subtract this value to one, and take $\log_p$ of the result. There have been various papers [4,5] describing approximations for calculating p^x and $\log_p x$. We adopt the invariant method [4] here. This method is used to compute exponentials and logarithms with base e in the original paper. Through some modifications, the method can be used also for other bases, such as 2. In the following, we'll describe how to use the invariant method to calculate $2^{**}x$ and $\log_2 x$.

The basic idea of the invariant method is to add one more independent variable and map the new function into a plane which is parallel to the x-y plane. The iteration is performed in such a way that the value of the new function remains constant. The pair (x,y) is computed iteratively from the starting point with given x and a proper initial value of y to the final point with y equal to the desired answer z and x equal to some fixed value. The invariance method can be illustrated as in Fig.2-1.

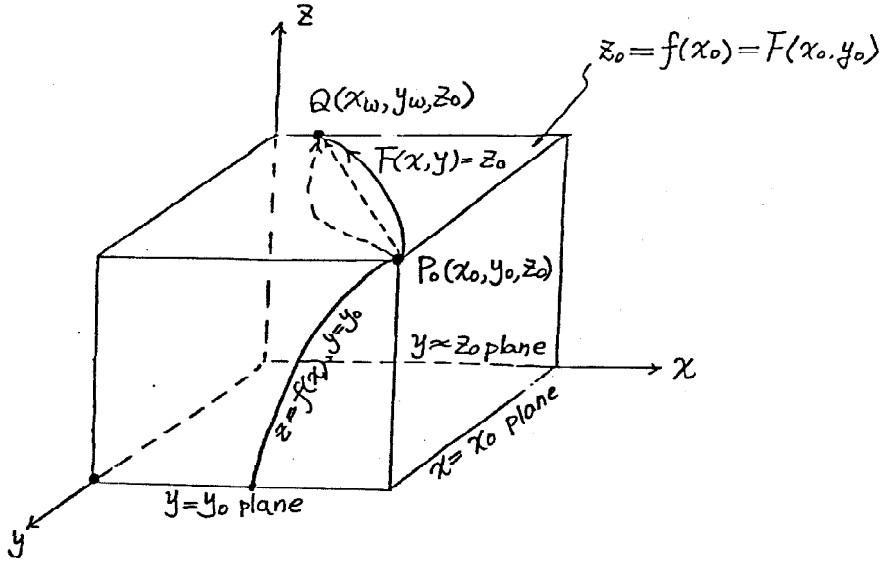


Fig.2-1. The invariant method of $f(x_0)$

There are an infinite number of curves connecting two fixed points (x_0,y_0), (x_w,y_w) in the plane $z=f(x_0)$=constant. The shortest path between two points is a straight line. But the more important consideration is to make the computation of each iteration step simple and straightforward in a hardware implementation. A set of iterative factors which has the form $\log_2(1+2^{-m})$, where m is an integer, is chosen and stored in a ROM table. By using this table, all the computations of iteration steps are just add/subtract, shift and table look-up. In the following, I'll explain how to use invariant method to solve a exponential function and a logrithmic function.

1). For a base 2 exponential function 2^x, where $-1 \leq x < 1$, we have

$$z_0 = f(x_0) = 2^{x_0} = y_0 * 2^{x_0}$$
$$= (y_0 * a_0) * 2^{**}(x_0 - \log_2 a_0) = y_1 * (2^{**} x_1)$$
$$= (y_1 * a_1) * 2^{**}(x_1 - \log_2 a_1) = y_2 * (2^{**} x_2)$$
$$= \ldots\ldots\ldots$$
$$= y_n * 2^{x^{**}n} = y_n * 2^0 = y_n$$

initialization	$y_0 = 1$
function	$F(x,y) = y * 2^x$
transformation	$x_{k+1} = x_k - \log_2 a_k$
	$y_{k+1} = y_k * a_k$
termination	$x_n = 0$, $z_0 = y_n$
transfer factor	$a_k = 1 + 2^{**}(-m)$ for $x_k > 0$
	$a_k = 1 - 2^{**}(-m)$ for $x_k < 0$

where m is the position of the first non-zero bit of $|x_k|$.

For each iteration, the first non-zero bit of x_k is removed and x_{k+1} is closer to 0.

A. For $x_k > 0$:

Assume $x_k=2^{**}(-m)+p$, $p<2^{**}(-m)$, i.e. the first (m-1) bits to the right of the decimal point are all zeros, the m-th bit is the first 1 bit of x_k.

then $a_k=1+2^{**}(-m)$ and

$$\begin{aligned} x_{k+1} &= x_k - \log_2 a_k \\ &= (2^{**}(-m)+p) - \log_2(1+2^{**}(-m)) \\ &= (2^{**}(-m)+p) - (1/\ln 2)*(2^{-m}-2^{-2m}/2+2^{-3m}/3-\ldots\ldots) \\ &= (1-(1/\ln 2))*2^{-m}+p+(1/\ln 2)(2^{-2m}/2-2^{-3m}/3+\ldots\ldots) \\ &= -0.43*2^{-m}\ p+O(2^{-2m}) \\ &< 2^{-m} \end{aligned}$$

x_{k+1} is less than 2^{-m}, so the '1' bit in 2^{-m} position is removed, but the first ter. in the righthand side of the function of x_{k+1} is a negative number which is sligthly less than $2^{-(m+1)}$, x_{k+1} can be positive or negative and it depends on p. When x_{k+1} overshoots to negative, the condition goes to the case B.

B. For $x_k<0$:

Assume $x_k=-(2^{-m}+p)$, $p<2^{-m}$, then let $a_k=1-2^{-m}$ and

$$\begin{aligned} x_{k+1} &= x_k - \log_2 a_k \\ &= -(2^{-m}+p) - \log_2(1-2^{-m}) \\ &= -(2^{-m}+p) - (1/\ln 2)*(-2^{-m}-2^{-2m}/2-2^{-3m}/3-\ldots\ldots) \\ &= 0.43*2^{-m}-p+O(2^{-2m}) \end{aligned}$$

Just similar as case A, the first non-zero bit of x_k is removed in x_{k+1}, and the sign of x_{k+1} depends on the value of p.

This algorithm provides x a zig-zag (not exactly) approach to the final value 0. We need two tables to store the values of $\log_2(1+2^{-m})$ and $\log_2(1-2^{-m})$. The number

of iterations is N for the worst case, where N is the word length of x. The length of each table is N. The table length can be reduced to N/2.

Since $\log_2(1\pm2^{-m})$

$=(1/\ln 2)*(\pm2^{-m}-2^{-2m}/2\pm2^{-3m}/3-.....)$

$\cong \pm(1/\ln 2)*2^{-m}$ when $m>N/2$

So, store only the values of $\log_2(1+2^{-m})$ when $m<N/2$. For $m>N/2$, shift the N/2-th term of the table right by (m-N/2) bits. It's easy to implement.

y_{k+1} is computed by shifting y_k right m positions and then adding or subtracting y_k from this value.

$$y_{k+1}=y_k*a_k$$
$$=y_k*(1\pm2^{-m})$$
$$=y_k\pm y_k*2^{-m}$$

2). For a base 2 logarithm $\log_2 x$, where $0<x\leq1$

$$z_0=f(x_0)=\log_2 x_0=y_0+\log_2 x_0$$
$$=(y_0-\log_2 a_0)+\log_2(x_0*a_0) =y_1+\log_2 x_1$$
$$=(y_1-\log_2 a_1)+\log_2(x_1*a_1) =y_2+\log_2 x_2$$
$$=.....$$
$$=y_n-\log_2 x =y_n-\log_2 1=y_r$$

Initialization:	$y_0=0$
Function	$F(x,y)=y+\log_2 x$
Transformation	$x_{k+1}=x_k*a_k$
	$y_{k+1}=y_k-\log_2 a_k$
Termination	$x_n=1,\ y_n=z_0$
Transition factor	$a_k=1+2^{-m}$

The transition factor is also selected as $a_k=1+2^{-m}$, where m is the first non-zero bit of $(1-x_k)$. Hence x_{k+1} will be closer to 0.

Assume $1-x_k=2^{-m}+p$, $p<2^{-m}$, and $a_k=1+2^{-m}$

then
$$\begin{aligned} x_{k+1}&=x_k*a_k \\ &=(1-2^{-m}-p)*(1+2^{-m}) \\ &=1-p-2^{-2m}-p*2^{-m} \\ &=1-p-O(2^{-2m}) \end{aligned}$$

so $1-x_{k+1}=p+O(2^{-2m}) < 2^{-m}$

x_{k+1} is computed by shifting and adding x_k . For the transformation of y_k,

$$\begin{aligned} y_{k+1}&=y_k-\log_2 a_k \\ &=y_k-\log_2(1+2^{-m}) \end{aligned}$$

A table of length N is used to store the value of $\log_2(1+2^{-m})$, where $1\leq m\leq N/2$. The number of iterations is N in the worst case. The actual iteration number depends on the bit pattern of x. So the average number of iterations is N/2. The table length can also be reduced to N/2 by providing the same shift scheme which is described at the previous section.

A scheme using the direct method to compute $\log_2(1+2^{-x})$, $x>0$

Step 1. Separate x into an integer part x_1 and a fractional part x_2

Step 2. Check x_1, if $x_1>N$, then stop.

Step 3. Compute 2^{-x_2} with the invariance method described above.

Step 4. Shift the result of (3) to the right by x_1 bits.

Step 5. Add the result of (4) to 1 and shift right 1 bit.

Step 6. Take the logarithm of the result of (5) by using the invariant method.

Step 7. Add 1 to the final result.

2.5. PERFORMANCE OF THE INVARIANT METHOD

In Table 1, there is a summary for the time, memory space and hardware required by each step of the invariant method.

Table 1. The time and hardware required for each step of the invariant method

	TIME	SPACE	HARDWARE
Step 1	-	-	registers
Step 2	-	-	switch
Step 3	[1](N+N/2)*(N/2)	N*(N/2)*2	two adders, one shifter
Step 4	N/2	-	shifter
Step 5	1	-	shifter
Step 6	[1](N+N/2)*(N/2)	N*N/2	two adders, one shifter
Step 7	N	-	one adder
Total	(3/2)*N*(N+1)	N*N	two adders, one shifter

(1. These are the average time. The maximal time spent is (N+N/2)*N for step 3, and (N+N/2)*((N/2)+1) for step 6.)

In table 2, the logarithm arithmetics are compared with the conventional floating-point arithmetic operations. In the view of hardware, a multiplier has the same dimension as a N**2 table, so the size of both two machines are almost the same. But in the view of processing time, the logarithmic unit is much worse than the floating-point unit for addition and subtraction, but a little better in division and almost the same in multiplication. Statistically, the addition occurs most frequently in the four operations. So in consideration of time, floating-point arithmetics is superior to the logarithm arithmetics.

Table 2. Comparison Between Logarithm and Floating-Point Arithmetics

	Time	Hardware
Addition/Subtraction		
1. F.P.	4*N	two adders,one shifter one counter
2. Log	2n+N*((3N/2)+7/2)	two adders, one shifter N^2 table
Multiplication		
1. F.P.	N	one multiplier,one adder
2. Log	N	one adder
Division		
1. F.P.	$2N*\log_2 N$	one multiplier,one adder
2. Log	N	one adder

III. FLOORPLAN AND BASIC CELLS

A chip has been designed and fabricated to implement and test the addition algorithm which was described before. To make it compatible with a double-precision floating-point processor, I choose 64-bit word length for the log number. On the other hand, to make its size suitable for a one-chip project. It is necessary to limit the data path to one bit in the current state of the technology. Hence, it was decided to design a serial machine. A serial machine also has other advantages over a parallel machine. It turns out that a serial machine can be designed with only local communication lines. Long, slow global wires that cause low clock speed of the whole machine can be avoided. The chip has two identical ALU's working in parallel. Associated with each ALU, there are four 64-bit and one 53-bit shift registers connected to the inputs and the outputs of the ALU and a PLA controller which takes the op code and the input/output register select code from the input pads and generates the microcode of the ALU and other control signals. In addition, a 6-bit programmable counter is used for system timing. Besides two separate processors, two ROM tables are shared by two processors for table look-up use. Each processor has a one-bit input/output port through which data can be shifted into or out of any one of the working registers. The floorplan is shown in Fig.3-1. Except for the ROM tables in the middle of the left side and the power and control lines, the rest of the design is up-down symmetrical. The processor located at the upper part in this chip is numbered by 1 and the one at the bottom is 2.

The size of this chip is 5138 x 6088 microns^2 with lambda=2.5 microns. The number of pads is 33. A two-phase nonoverlapping clock scheme is used for system timing. Input and output is assumed to be synchronous with in-chip clock.

Floorplan

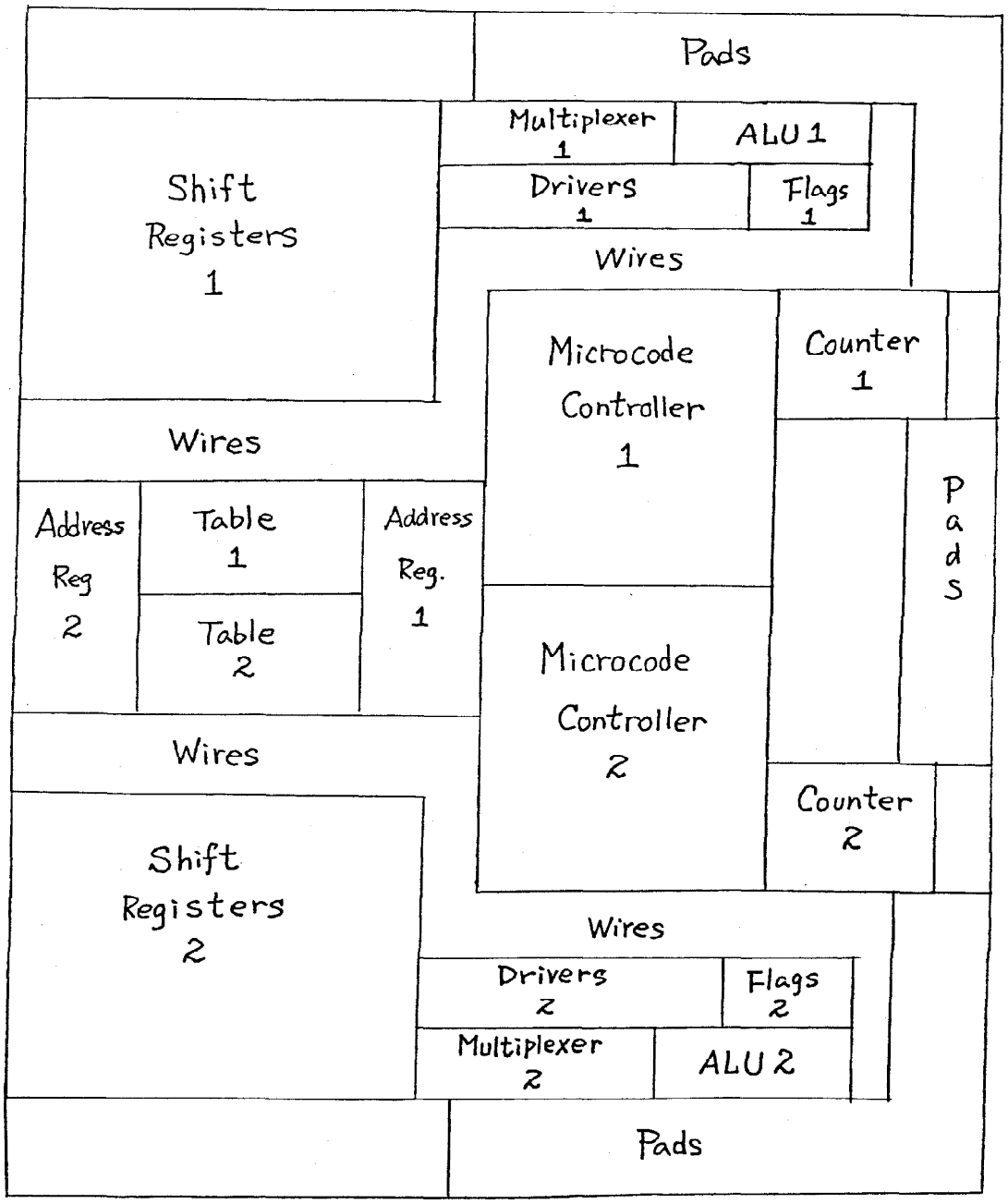


Fig. 3-1 The Floorplan of the Log Machine

1. ALU

This is a one-bit ALU, which is adapted from the bitslice of OM2 [6]. The ALU consists of the input latches, three functional blocks (carry propogate, carry kill and run blocks) and the output latch. The size is about 230 x 75λ^2.

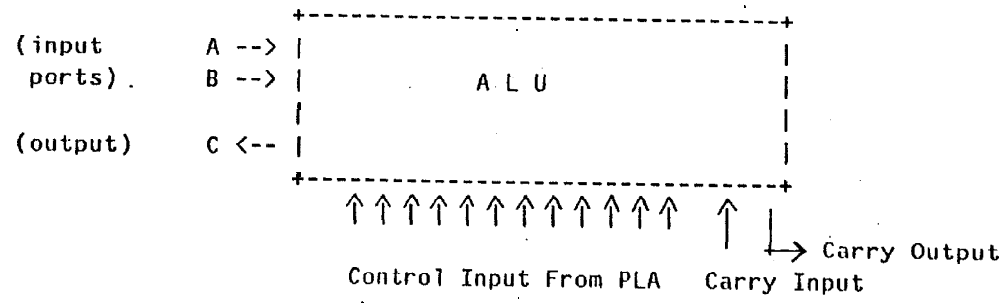


Fig. 3-2 The One-Bit ALU

The ALU takes 12 control lines (four for each functional block) and two carry input selection lines which are generated by the PLA controller. The ALU has two input ports and one output port which are connected to the outputs of the multiplexer. The inputs come into the ALU and C_{in} is precharged at clock phase 1. The control lines of three functional blocks is valid and the ALU operates during phase 2.

There are three one-bit flags associated with the ALU. The values of the flags are selected by three flag selection lines which come out from the PLA controller. The flag values will feed back to the input of the PLA controller and effect the next state transition.

2. Shift Registers

There are four 64-bit shift registers and one 53-bit shift register for each ALU. They are named as R[0:4]. The shift register set is located to the left of the ALU. The area is about 760 x 570λ^2.

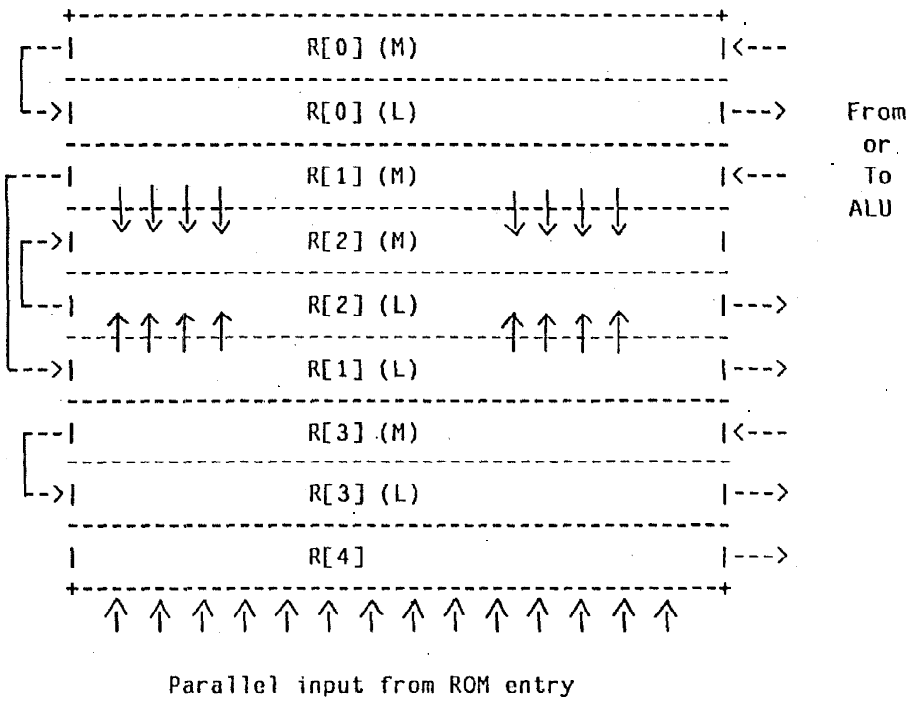


Fig. 3-3 The Shift Register Set

R[0],R[3] -- 64-bit shift registers with refresh loop. When a shift enable signal passes in, the data in the registers shift one bit right. Otherwise, each shift cell remains in the same state through the refresh loop. The register is folded into two parts so that the input and the output can be taken out in the same direction.

R[1],R[2] -- 64-bit shift registers with parallel loading and bit-reversal feature. R[1] is a serial-in, parallel- or serial-out shift register. R[2] is a parallel-in, serial-out shift register. R[2] is imbedded into R[1] and R[1] can load data to R[2] in parallel under the proper selection signal occurring during phase 1. R[1] has a refresh feature but R[2] does not. The data-flow direction of R[2] is opposite to that of R[1], so that R[2] is in bit-reversed order after being loaded from R[1]. This pair of shift registers is used to test the first nonzero bit of x(k) during the iteration steps in the invariance algorithm.

R[4] -- 53-bit parallel-in, serial-out shift register. This register is used to load the data of the ROM table. Since all the values stored in the table are less than 1, the

word length of the table is just 52 (the fractional part of 64-bit logarithmic representation). The MSB of R[4] is always connected to the V_{dd} or ground. Which connection depends on the sign of the value being loaded into R4. The MSB of R[4] is a sign-extension bit. Contrary to the other registers, R[4] is not folded and can only be loaded in parallel by the value of some table entry. There is no path to serially shift data into R[4].

R[0], R[1] and R[2] are connected to the upper input port (Input A) of the ALU. R[3] and R[4] are connected to the lower input port (Input B) of the ALU. Only R[0],R[1] and R[3] are connected to the output of the ALU. Between the shift registers and the I/O of the ALU, there are some very simple multiplexing circuits.

Any one of these five registers can be the source register of the output port but only one of R[0], R[1] and R[3] can be the destination register of the input port since R[2] and R[4] are constructed and used for special purposes only. They never receive data from outside.

3. ROM

Two 26-word long, 52-bit wide tables combined together form the ROM cell. Each table is the OR plane of an ordinary PLA. Table 1 stores the values of $-\log_2(1+2^{-m})$, where $0<m\leq 26$ and table 2 stores the values of $-\log_2(1-2^{-m})$, $0<m<26$. The negative numbers are represented in 2's complement. The words in both tables are fixed-point binary numbers with the decimal point located to the left of the MSB. The address of each table is a 26-bit serial-in, parallel-out shift register. The selection line comes out from the super buffer connected to each bit of the shift register. The size of the ROM cell is about $420 \times 420\lambda^2$. One shift register with its super buffers is about $145 \times 420\lambda^2$. The address cell is twice as wide as the table. Hence, the address cells are put at both sides of the ROM cell and the total area including the ROM tables, the address cells and the routing space is about $950 \times 450\lambda^2$. The data stored in the tables can be taken out from both the top and the bottom of the table so they can load data into the R[4]'s of both ALU's.

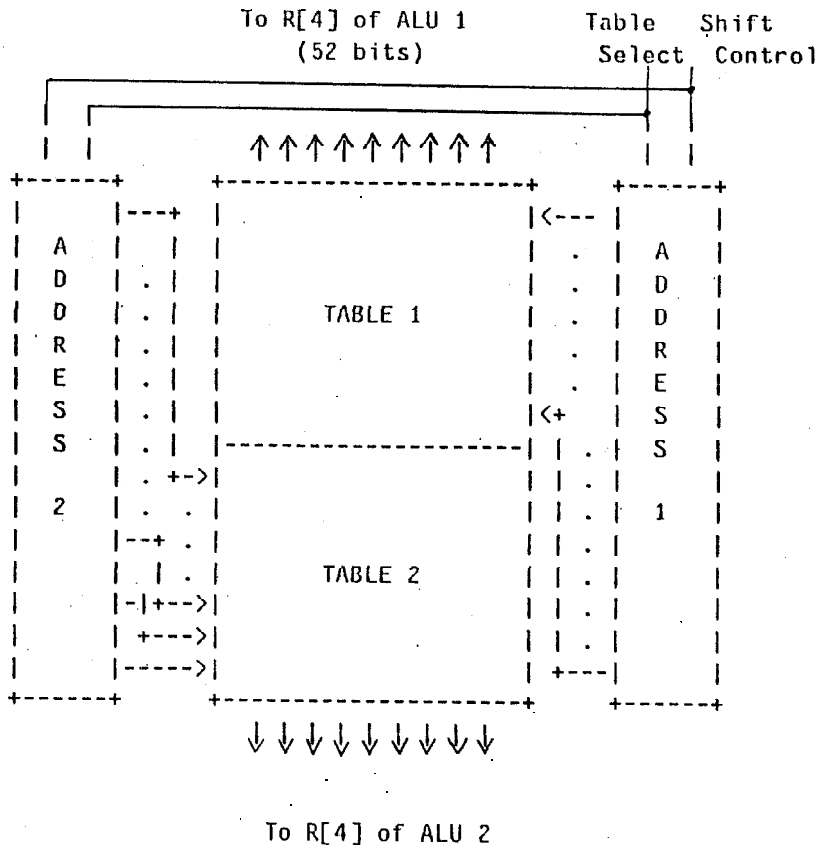


Fig. 3-4 Tables with Address Registers

4. Counter

The core of system timing is a six-bit programmable counter associated with a 5-term ROM cell and a 6-bit shift register C_R. Each bit of this counter is composed of a carry generating part (AND gate) and a sum generating part (EXOR gate). The counter can be preset to any initial value which is stored either in the ROM cell or in the shift register. It is used to control the state transition of the PLA controller. In most cases, the counter counts to 64 and then sends a signal to the PLA to accept the next op code and change the control signals. But, on doing addition or subtraction, the timing is varied by presetting the initial values of the counter. The counter can choose 1, 52, 12, 6 and 64 from ROM as a fixed initial value. It can also choose an arbitrary value from the shift register C_R which is loaded by the output of

the ALU and used later as a count number for output alignment in the invariance algorithm.

The size of the counter is about 250 x 90λ^2. The sum of the ROM cell and the 6-bit register cell is about 240 x 100λ^2; the total area of the counter with storage elements and the multiplexer is 250 x 220λ^2. The block diagram is shown in Fig. 3-5.

The carry bit of the MSB is taken out from the left side of the counter. Three counter selection lines coming from the output of the PLA select the initial value of the counter. The data of the shift register is taken directly from the output of the ALU when the flag F[2] is set.

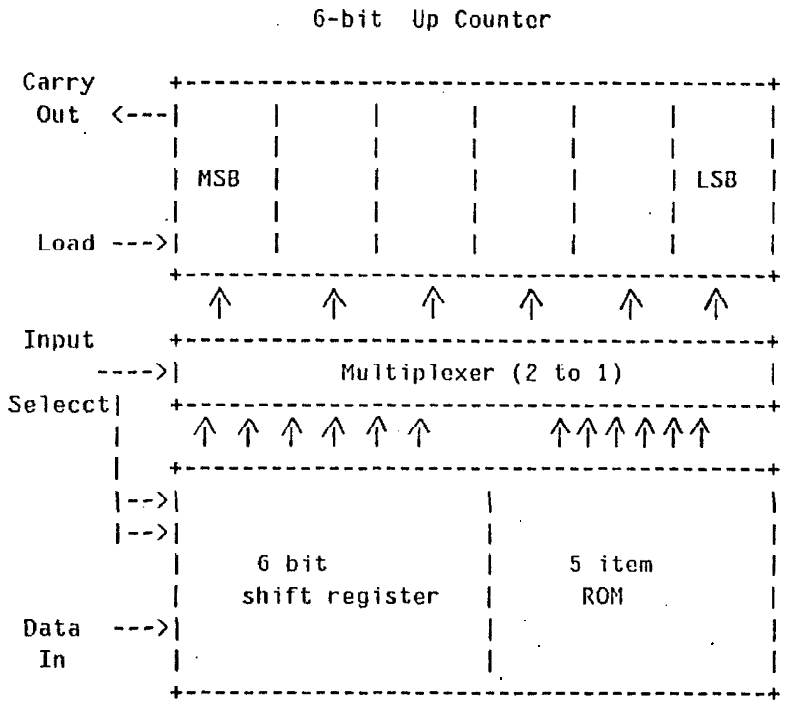


Fig. 3-5 Counter and Its Input Circuit

5. Drivers

There are two different kinds of drivers; one is the drivers of the ALU control lines and the other is the drivers of the input/output selection lines. All the inputs of the drivers come out from the PLA controller during phase 1. The outputs of the control drivers have to be valid at phase 2, so that there is a refresh loop during phase 2.

The I/O selection lines will be fed to all cells of all shift registers. Hence, a big driver is necessary. The input selection signals are ANDed with phase 1 of the clock and the output selection signals are ANDed with phase 2 of the clock to guarantee that the correct data will pass to the ALU.

6. PLA Controller

The PLA has 20 inputs, 35 outputs and 78 minterms. The input consists of a 5-bit op code, 5-bit input selection code, 3-bit output selection code from the input pads, a 4-bit sequence control code feedback from the output of itself, and 3 flags. It generates 12-bit ALU control signals, 5-bit input selection signals, 3-bit output selection signals, 4-bit sequence control code used to count how many state transitions is needed for one instruction, 3-bit counter initial value selection code, 2-bit C_{in} selection code, 3-bit flag selection code and three other selection signals; one for table look-up, one for bit-reversal of R[1] and R[2] , the other is for table address selection. The microcodes and the function of all the input and output lines of this PLA is described in App.2.

The inputs are latched into the PLA during phase 2 and the output is valid during phase 1. Only when a control signal which is generated by the counter is received, can the outputs of the PLA be latched into the drivers. The size of the PLA is about $590 \times 680\lambda^2$. It is the largest cell in the whole chip.

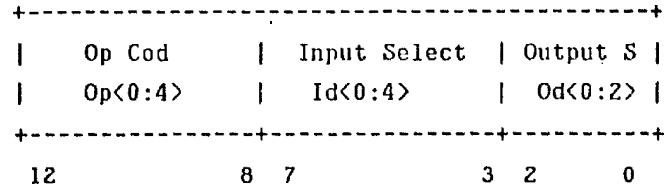
IV. INSTRUCTION SET

At this moment, the instruction set can only handle the arithmetic operations, logic functions and very simple I/O commands. The instruction set is sufficient to run and test the invariance algorithm. The instruction set has no provisions for memory access (I don't have any memory anyway.), and sequence control; thus, it is highly restricted.

The instruction set has 13-bit fixed-length format. The instruction format and the list of op codes are as follows:

Instruction Format

I_{op}<12:0>



Op Code Description

No	Nem	Op<0:4>	Description
0	Nop	00000	no operation
1	Clear	00001	C <-- 0
2	MoveA	00010	C <-- A
3	MoveB	00011	C <-- B
4	AND	00100	C <-- A $\wedge$ B
5	OR	00101	C <-- A $\vee$ B
6	EXOR	00110	C <-- A $\oplus$ B
7	Mult	00111	C <-- A + B
8	DivA	01000	C <-- A - B
9	DivB	01001	C <-- B - A
10	IncA	01010	C <-- A + 1
11	IncB	01011	C <-- B + 1
12	DecA	01100	C <-- A - 1
13	DecB	01101	C <-- B - 1
14	Set1	01110	C <-- 1.0
15	Set	01111	C <-- $111..1_2$
16	NegA	10000	C <-- -A
17	NegB	10001	C <-- -B
18	ShrA	10010	A <-- Cin ∏ shr(A)
19	ShrB	10011	B <-- Cin ∏ shr(B)
20	NOT-A	10100	C <-- $\neg$A
21	NOT-B	10101	C <-- $\neg$ B
22	Add	10110	C <-- $\log_2(2^A+2^B)$
23	Sub	10111	C <-- $\log_2(2^A-2^B)$
24	IN	11000	Input
25	OUT	11001	Output

Note: 1. All instructions have the same format, no matter what the binary or unary operation is.

2. A and B are the two input registers of the ALU and C is the destination register of the ALU. A, B and C are chosen by the input selection code Id and the output selection code Od.

3. In this instruction set, destination register C is not necessary to be one of two source registers.

4. ShrA and ShrB are one-cycle-time instructions and the destination register must be the source register. All the other instructions, except Add and Sub, take 64 cycle times (i.e. one word time) to execute.

Input Selection

Id<0:4>	Register	Connection
00001	R[0]	A
00010	R[1]	A
00100	R[2]	A
01000	R[3]	B
10000	R[4]	B

Note: The first three registers R[0,1,2] are connected to the ALU input port A and the other two R[3,4] to port B. In binary operation, one of R[0:2] and one of R[3,4] have to be chosen at the same time, i.e. the Id<0:4> may have the form 10010,.., etc. In unary operation, only one of five registers will be chosen, depending on which port is involved in that instruction. In some other instructions, such as Clear, Set, Setl, neither of two input ports is needed so that the Id becomes 00000. The code is alse used to select a source register for the OUT instruction.

Output Selection

Od<0:2>	Register
001	R[0]
010	R[1]
100	R[3]

Note 1: R[0], R[1] and R[3] are connected to the ALU output port C. Any one or any pair of them can be selected as output in one instruction. The code is also used to select a destination register for the IN instruction. R[2] and R[4] are not connected to the ALU output port because both of them are parallel-in, serial-out shift register. They can't take serial input at all.

2: "Output Selection" means "select a register as the output of the ALU". Actually, one datum is shifted "into" the selected register. Thus, at instruction "IN", data is read from the external world into one of the three registers chosen by Od<0:2>. On the contrary, at instruction "OUT", data is output to outside from the register which is chosen by Id<0:4>.

V. TIMING

A two-phase, nonoverlapping clock scheme is used as the system clock in this chip. The log-machine is a strictly synchronous machine. By the assumption of serial machine, there are no global wires, no big drivers, thus the machine speed should be very fast. But the PLA controller in the chip is not a serial cell. It becomes the slowest part in the chip. The estimated propogation delay of the PLA is about [7]

$$\tau\,[\,5xM + 4.3xN + 4.6xI + 4]$$

where M is the number of minterms
N is the number of inputs
I is the number of outputs
τ is the process transit time

In my chip, the delay is about 190 ns, assuming a transit time τ of 0.3 ns.

The PLA is operated during phase 1 of the clock. Hence, the duration of the phase 1 must be longer than 190 ns. The duration of phase 1 is estimated to about 40ns ($\approx 130\tau$) according to the models in Mead & Conway [8]. Each long control line going into the shift register cells drives more than 60 transistors. The wire delay of those control lines is equivalent to 300 minimal gate delays. Since the super buffer drivers are four times as big as the minimal one, the delay introduced by the control lines is about 90τ. The typical delay of the inverter plus pass transistor is about 30τ. If an other 8τ inverter delays are added for the drivers, the total time of phase 1 is about 130τ(40 ns).

By the estimation made in the last paragraph, the minimal clock period is 230 ns. So, the average addition or subtraction time is about 920μs (0.92 ms) and the multiplication and division time is about 15μs.

VI. SIMULATION

This simulation program is written in Simula. The shift register is simulated by a Simula CLASS register. Each instruction is a procedure within CLASS register, since each of the instructions takes one or two registers as input and return another register as output. Two tables are simulated by a SUBCLASS table of CLASS register. A table is an array of registers and the address register of the table is also a shift register. Table has its own operation -- lookup and table is initiated to the values of

table entries as soon as it is constructed.

The purpose of this simulator is to test the correctness and performance of the invariance algorithm. The log-machine takes logarithmic numbers as input and output so that some auxilliary procedures are needed to convert decimal numbers to binary logarithmic numbers and vice versa. Two global variables, count and clock are used to count the number of iteration steps and the number of clock cycles in the addition/subtraction algorithm. The algorithm is suitable for hardware implementation but is not sufficient on a conventional machine. Hence, the execution time of the simulation program is not a meaningful figure.

The result of some sample data is shown in App.3. Compared with the answer derived by DEC-20 double-precision logarithmic algorithm, the two answers are totally identical within a precision of 10^{-13}. The number of iterations and the total clock cycle are just as predicted. The former number is in the range of O(N) and the latter one is in the range of $O(N^2)$. A summary of those data is shown in Table 1.

The single precision floating point number of the DEC 20 has a 27-bit mantissa part. Its precision is equivalent to the precision of a 39-bit logarithmic number containing a 12-bit integer part and a 27-bit fractional part. Using this representation, it turns out that the invariant method is more precise than the DEC 20 single precision lgarithmic algorithm in the computation of $\log_2(A+B)$. Some results are shown in Table 2.

The error generated by the invariance method is in the order of 2^{-N}, where N is the length of the fractional part. The simulator simulates the invariance method under different precision. Three sets of results are shown in Table 3 (A),(B) and (C). The error increases exponentially with decreasing word length. Figure 5-1 shows the relation between the word length and the errors. It is reasonably matched with the theoretical prediction.

Table 1

Data A	Data B	# Iterations	# Clocks
2.0	4.0	30	3313
2.0	6.0	14	1630
4.0	12.0	13	1501
40.0	216.0	40	4442
92.0	8100.0	46	5032
4002.0	4190.0	41	4545
1024.0	7166.0	51	5591
6.667	9.333	46	5040
90.4556	112.3547	40	4431
Average		35.7	3947.2
Theoretical		52	4262

Table 2

			Errors	
Data A	Data B	# Iterations	Invariance Method	Single Precesion (DEC-20)
2.0	4.0	16	6.05×10^{-9}	1.63×10^{-8}
2.0	6.0	11	1.49×10^{-8}	2.98×10^{-8}
4.0	12.0	11	1.50×10^{-8}	5.96×10^{-8}
40.0	216.0	21	5.20×10^{-8}	1.19×10^{-7}
92.0	8100.0	25	$< 10^{-15}$	1.19×10^{-7}
4002.0	4190.0	20	7.45×10^{-9}	1.19×10^{-7}
1024.0	7166.0	23	3.72×10^{-9}	2.38×10^{-8}
6.667	9.333	22	2.98×10^{-8}	5.96×10^{-8}
90.4556	112.3547	19	5.65×10^{-9}	4.65×10^{-8}

Table 3 (A)

The Result of Different Precision

Data A 2.0

Data B 6.0

Precise Answer 3.0

Word Length	# Iterations	# Clocks	Errors
64	14	1630	1×10^{-15}
60	14	1574	1.8×10^{-14}
56	14	1518	2.3×10^{-13}
52	14	1462	4.5×10^{-12}
48	14	1406	7.3×10^{-11}
44	13	1239	4.7×10^{-10}
40	12	1099	1.12×10^{-8}

Table 3 (B)

The Result of Different Precision

Data A 90.4556

Data B 112.3547

Precise Answer 7.6639871132283

Word Length	# Iterations	# Clocks	Errors
64	40	4431	4×10^{-15}
60	35	3719	4.7×10^{-14}
56	36	3752	1.1×10^{-12}
52	34	3434	1.6×10^{-11}
48	30	2906	2.1×10^{-10}
44	25	2344	1.46×10^{-9}
40	23	2065	5.65×10^{-9}

Table 3 (C)

The Result of Different Precision

Data A 92.0

Data B 8100.0

Precise Answer 1 3.0

Word Length	# Iterations	# Clocks	Errors
64	46	5032	1×10^{-14}
60	44	4699	1.1×10^{-13}
56	35	3553	4.5×10^{-13}
52	34	3371	1.2×10^{-11}
48	31	2966	1.8×10^{-10}
44	29	2730	2.1×10^{-9}
40	25	2270	2.24×10^{-8}

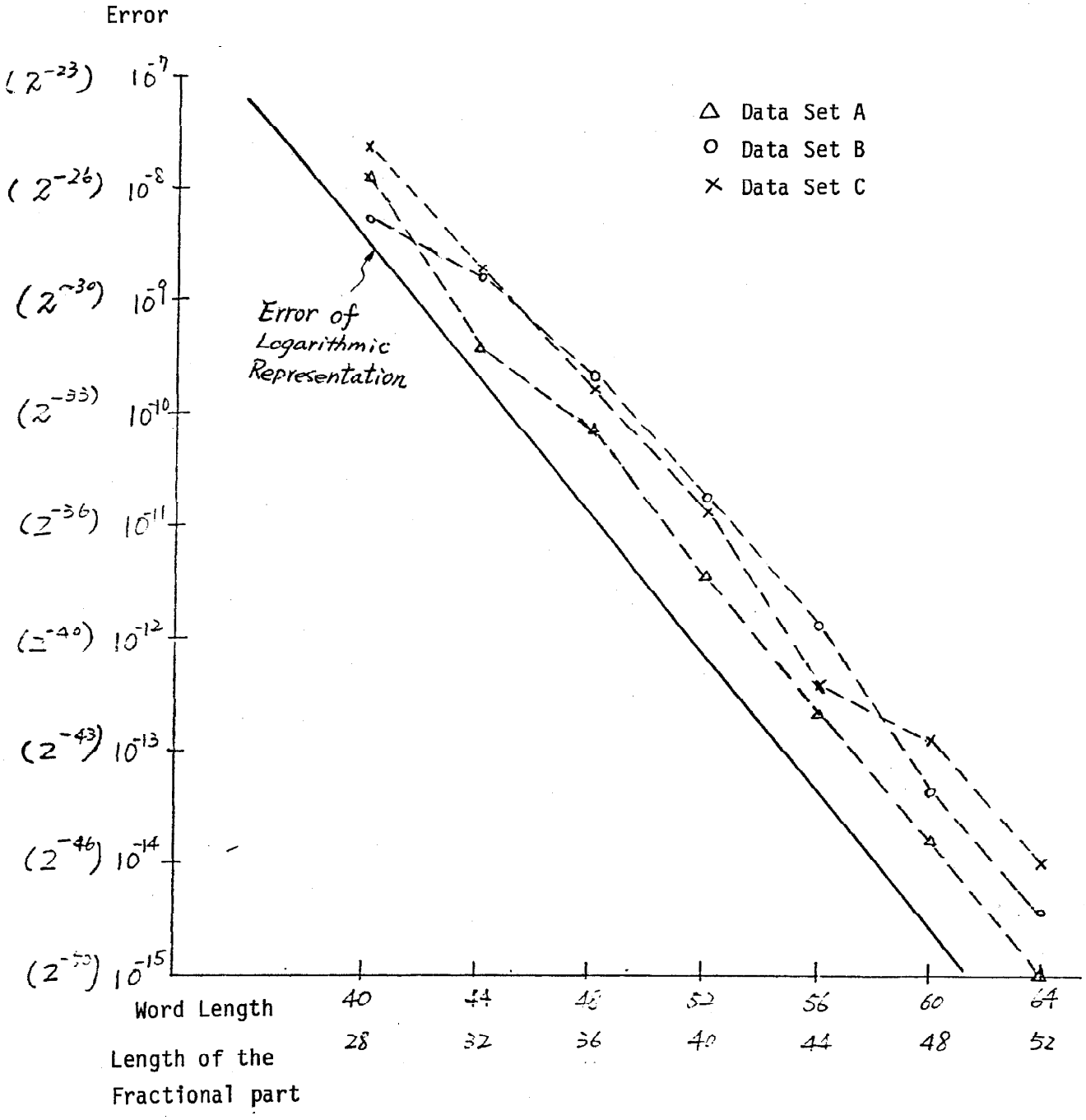


Fig. 6-1 The Error of the Invariant Method VS. the Word Length

VII. CONCLUSION

The logarithmic number representation is a potential alternative to the floating point representation. The invariance method for the logarithmic addition/subtraction is a good algorithm applying to the numbers with arbitrary precision. It is especially suitable for serial arithmetics because it has simple add and shift operations. From the last section, the average time for the addition and subtraction is much worse than any existing floating point processor. One of the reasons is that the log machine is a serial machine. If we build a parallel machine with the same clock cycle, the operation time is quite good compared to any other floating point processor. The second reason is that we can't avoid the parallel path in the chip. The two slowest cells are the PLA controller and the control lines; both have parallel features. If we can remove the PLA controller from the chip and allow more than one cycle time to fetch intructions, then we can significantly increase the speed of the chip. The cycle time can be reduced to 80 ns and the ADD/SUB instruction time is also reduced to about 320μs.

If we can avoid all the long, global wires in the chip, then the speed of the chip will be very fast. The clock cycle will be in the range of 10ns. Then, this tiny, one-chip, serial log-machine can compete with any existing floating point processors. But, can we get rid of those global wires in a serial processor? It is still an open question. It is challenging work to find the answer.

REFERENCE

(1). N.G. Kingsburg and P.J. Rayner,"Digital filtering using logarithmic arithmetic" Electron. Lett., vol.7, pp.56-58, Jan. 28, 1971.

(2). E.E. Swartzlander, Jr. and A.G. Alexopoulos, "The sign/logarithm number system" IEEE Trans. Comput., vol. C-24, pp.1238-1243, Dec. 1975.

(3). S.C. Lee and A.D. Edgar, "The focus number system" IEEE Trans. Comput., vol. C-26, pp.1167-1170, Nov. 1977.

(4). T.C. Chen, "Automatic computation of exponentials, logarithms, ratios and square roots" IBM J. Res. Develop., pp.380-388, July 1972.

(5). W.H. Specker, "A class of algorithms for ln x, exp x, sin x, cos x, tan x, and cot x" IEEE Trans. Electronic Computers, EC-14, pp.85-86, 1965.

(6). Mead & Conway, Introduction to VLSI System, Chapter 5, 1980, Addison-Wesley

(7). Jim Cherry, 'PLA Speed Calculations', July 1979

(8). Mead & Conway, Introduction to VLSI System, Chapter 1-13, 1980, Addison-Wesley

APPENDIX 1

ISP description for LOG-MACHINE

Internal State

```
!Upper Half;
  R_a1[0:3]<63:0>      !Shift Registers with 64 bits per word
  R_a1[4]<52:0>        !Shift Register with 53 bits per word,
                            the MSB is sign-extension bit.
  F_1<1:3>             !Three flags
  C_count1<6:0>          !Sequence Up-Counter, 6 bit with one carry.
  C_R1<5:0>            !Temp Register which feeds input of counter.

!Lower Half;
  R_a2[0:3]<63:0>
  R_a2[4]<52:0>
  F_2<1:3>
  C_count2<6:0>
  C_R2<5:0>

!Shared Tables;
  M_T1[1:26]<51:0>     !Table 1, 26 words with 52 bits per word
  M_T2[1:26]<51:0>     !Table 2,
  A_M1<1:26>           !Address Register of Table 1
  A_M2<1:26>           !Address Register of Table 2
```

Interface State

```
  I_op1<12:0>          !Instruction code input for ALU 1
  IO_d1<0>             !Data Input-Output pad for ALU 1
  E_d1<0>              !Enable line output to synchronize I/O of ALU 1

  I_op2<12:0>
  IO_d2<0>
  E_d2<0>
```

Combinational Expressions

```
Op<0:4> := Ip<12:8>    !op code
Id<0:4> := Ip<7:3>     !Input register selection
Od<0:2> := Ip<2:0>     !Output register selection
S:=C_count<6>          !Singal to change state
```

Sequential Equations

```
execute-instruction := (S --> (
   nop    (:= Op=0) --> C_count <-- 63_10,
   clear  (:= Op=1) --> C_count <-- 0, Output <-- 0; repClear,
   moveA  (:= Op=2) --> Output <-- InputA, C_count <-- 0; repMA,
   moveB  (:= Op=3) --> Output <-- InputB, C_count <-- 0; repMB,
   AND    (:= Op=4) --> Output <-- InputA ∧ InputB,
                        C_count <-- 0; repAND,
   OR     (:= Op=5) --> Output <-- InputA ∨ InputB,
                        C_count <-- 0; repOR,
   EXOR   (:= Op=6) --> Output <-- InputA.EXOR.InputB,
                        C_count <-- 0; repEXOR,
   mult   (:= Op=7) --> Cout <-- Carry(InputA + InputB + Cin),
                        Output <-- Sum(InputA + InputB + Cin),
                        C_count <-- 0; repMult; F[1] <-- Cout,
   divA   (:= Op=8) --> Cout <-- Borrow(InputA - InputB - Cin),
                        Output <-- Diff(InputA -InputB - Cin),
                        C_count <-- 0; repDivA; F[1] <-- Cout,
   divB   (:= Op=9) --> Cout <-- Borrow(InputB - InputA - Cin),
                        Output <-- Dif(InputB -InputA - Cin),
                        C_count <-- 0; repDivB; F[1] <-- Cout,
   IncA (:= Op=10) --> C_count <-- 12, Output <-- 0; repClear;
                        C_count <-- 52, Cin <-- 1,
                        Cout <-- InputA ∧ Cin,
                        Output <-- InputA.EXOR.Cin; repIncA,
   IncB (:= Op=11) --> C_count <-- 12, Output <-- 0; repClear;
                        C_count <-- 52, Cin <-- 1,
                        Cout <-- InputB ∧ Cin,
                        Output <-- InputB.EXOR.Cin; repIncB,
   DecA (:= Op=12) --> C_count <-- 12, Output <-- 0; repClear;
```

```
                        C_count <-- 52, Cin <-- 1,
                        Cout <-- ¬InputA ∧ Cin,
                        Output <-- InputA.EXOR.Cin; repDecA,
DecB (:= Op=13) --> C_count <-- 12, Output <-- 0; repClear;
                        C_count <-- 52, Cin <-- 1,
                        Cout <-- ¬InputB ∧ Cin,
                        Output <-- InputB.EXOR.Cin; repDecB,
Setl (:= Op=14) --> C_count <-- 12, Output <-- 0; repClear;
                        C_count <-- 52, Cin <-- 1,
                        Output <-- Cin, Cout <-- 0; repSetl,
NegA (:= Op=16) --> C_count <-- 0, Cin <-- 1,
                        Cout <-- ¬InputA ∧ Cin,
                        Output <-- ¬InputA.EXOR.Cin; repNegA,
NegB (:= Op=17) --> C_count <-- 0, Cin <-- 1,
                        Cout <--  InputB   Cin,
                        Output <--   InputB.EXOR.Cin; repNegA,
Set  (:= Op=18) --> C_count <-- 0, Output <-- 1; repSet,
ShR  (:= Op=19) --> C_count <-- 63, Output <-- Cin; shift,
NOT-A(:= Op=20) --> C_count <-- 0, Output <-- ¬InputA; repNOTA,
NOT-B(:= Op=21) --> C_count <-- 0, Output <-- ¬InputB; repNOTB,
Add  (:= Op=22) --> C_count <-- 0, R[1] <-- InputA - InputB,
                        F[3] <-- Cout, SCo <--1111;
  (F[3] --> nextstep (:= (
       SCo=1111 --> C_count <-- 12,
                        R[1]<63:0> <-- R[1]<51:0>▯ R[1]<63:52>,
                        SCo <-- 1110; nextstep,
       SCo=1110 --> C_count <-- 52,
                        R[1]<63:0> <-- 111..1▯ R[1]<63:12>,
                        C_RI<5:0> <-- R[1]<5:0>, SCo <-- 1101; nextstep,
       SCo=1101 --> C_count <-- 63, E<1> <-- 1,
                        R[2]<0:63> <-- R[1]<63:0>, SCo <-- 1110; nextstep,
       SCo=1100 --> C_count <-- 52, F[1] <-- R[2]<0>,
                        R[2]<63:0> <-- R[2]<11:0>▯ R[2]<63:12>,
                        SCo <-- 1011; nextstep,
       SCo=1011 --> C_count <-- 12,
                        loop (:=( F[2] -->F[2] <-- F[1].EXOR.R[2]<0>,
                                               F[1] <-- R[2]<0>;
                                               E<3> <-- 1,
                                               A_M1<1:26> <-- 0▯ A_M1<1:25>
                                               A_M2<1:26> <-- 0▯ A_M2<1:25>
```

```
                                        loop,)),
                    SCo <-- 1010; nextstep,
          SCo=1010 --> (F[2] --> C_count <-- 63, E<2> <--1,
                                (F[1] --> R[4] <-- T_a1[A_M1],
                                  F[1] --> R[4] <-- T_a2[A_M2]),
                                SCo <-- 1001; nextstep,
                        ¬F[2] --> C_count <-- C_R<5:0>, nop,
                                  SCo <-- 0111; nextstep,)
          SCo=1001 --> C_count <-- 0, R[1] <-- R[1] + R[4],
                    SCo <-- 1101; nextstep,
          SCo=0111 --> C_count <-- 0, R[1] <-- 0,SCo <-- 0111; nextstep,
          SCo=0110 --> C_count <-- 63, nop,SCo <-- 0101; nextstep,
          SCo=0101 --> C_count <-- 12, nop,SCo <-- 0100; nextstep,
          SCo=0100 --> (F[2] --> C_count <-- 63, E<3> <-- 1,
                                  R[4] <-- T_a1[A_M1],
                                  SCo <-- 0011; nextstep,
                        ¬F[2] --> C_count <-- 12,
                                  R[1] <-- R[1]<51:0>▯R[1]<63:52>,
                                  SCo <-- 0010; nextstep,)
          SCo=0011 --> C_count <-- 0, R[1] <-- R[1] + R[4],
                    SCo <-- 0110; nextstep,
          SCo=0010 --> C_count <-- 52, R[1] <-- R[1] + 1,
                    SCo <-- 0001; nextstep,
          SCo=0001 --> C_count <-- 0, R[3] <-- R[1] + R[3],
                    SCo <-- 0000; execute-instruction,)),
¬F[3] --> next (:= (
          SCo=1111 --> C_count <-- 12, R[1] <-- 111..▯R[1]<63:52>,
                    R[3] <-- 111...▯R[3]<63:52>,
                    SCo <-- 1110; next,
          SCo=1110 --> C_count <-- 52, R[1] <-- 111...▯R[1]<63:12>,
                    R[3] <-- 111...▯R[3]<63:12>,
                    C_R<5:0> <-- R[1]<5:0>,
                    SCo <-- 1101; next,
          SCo=1101 --> C_count <-- 63, nop, SCo <-- 1100; next,
          SCo=1100 --> C_count <-- 52, nop, SCo <-- 1011; next,
          SCo=1011 --> C_count <-- 12, R[3]<63:0> <-- 0▯R[3]<63:1>,
                    SCo <-- 1010; next,
          SCo=1010 --> (F[2] --> C_count <-- 63, nop,
                                  SCo <-- 1001; next,
                        ¬F[2] --> C_count <-- C_R<5:0>,
```

```
                                R[1] <-- 0▯ R[1]<63:1>,
                                SCo <-- 1000, next,)
          SCo=1001 --> C_count <-- 0, (
                        F[1] --> R[1] <-- R[1] + R[3],
                        F[1] --> R[1] <-- R[1] - R[3],),
                       SCo <-- 1101; next,
          SCo=1000 --> C_count <-- 12,
                       R[1],R[3] <-- R[1]<51:0>▯ R[1]<63:52>,
                       SCo <-- 0111; next,
          SCo=0111 --> C_count <-- 52, R[1],R[3] <-- R[1] +1,
                       SCo <-- 0110; next,
          SCo=0110 --> C_count <-- 63, R[1],R[3] <-- 0▯ R[1]<63:1>,
                       SCo <-- 0101; next,
          SCo=0101 --> C_count <-- 63, E<1> <-- 1,
                       R[2]<0:63> <-- R[1]<63:0>,
                       SCo <-- 0100; next,
          SCo=0100 --> C_count <-- 52,
                       R[2]<63:0> <-- R[2]<11:0>▯R[2]<63:12>,
                       SCo <-- 0011; next,
          SCo=0011 --> C_count <-- 12,
                       R[2] <-- R[2]<0>▯ R[2]<63:1>, F[2] <-- R[2]<0>,
                       R[3] <-- 0 ▯R[3]<63:1>, E<3> <-- 1,
                       A_MI<1:26> <-- 0  A_MI<1:25>,
                       SCo <-- 0010; next,
          SCo=0010 --> (F[2] --> C_count <-- 63, nop,
                                 SCo <-- 0001; next,
                        ¬F[2] --> C_count <-- 0, nop,
                                  SCo <-- 0000; execute-instruction,)
          SCo=0001 --> C_count <-- 0, R[1],R[3] <-- R[1] + R[3],
                       SCo <-- 0101; next,))),
Sub  (:= Op=22) --> C_count <-- 0, R[1] <-- InputA - InputB,
                    F[3] <-- Cout, SCo <--1111;
  (F[3] --> nextstep (:= (
          SCo=1111 --> C_count <-- 12,
                       R[1]<63:0> <-- R[1]<51:0>▯ R[1]<63:52>,
                       SCo <-- 1110; nextstep,
          SCo=1110 --> C_count <-- 52,
                       R[1]<63:0> <-- 111..1▯ R[1]<63:12>,
                       C_RI<5:0> <-- R[1]<5:0>, SCo <-- 1101; nextstep,
          SCo=1101 --> C_count <-- 63, E<1> <-- 1,
```

```
                R[2]<0:63> <-- R[1]<63:0>, SCo <-- 1110; nextstep,
        SCo=1100 --> C_count <-- 52, F[1] <-- R[2]<0>,
                R[2]<63:0> <-- R[2]<11:0>▯ R[2]<63:12>,
                SCo <-- 1011; nextstep,
        SCo=1011 --> C_count <-- 12,
                loop (:= (F[2] -->F[2] <-- F[1].EXOR.R[2]<0>,
                                F[1] <-- R[2]<0>;
                                E<3> <-- 1,
                                A_M1<1:26> <-- 0▯A_M1<1:25>
                                A_M2<1:26> <-- 0▯A_M2<1:25>
                                loop,)),
                SCo <-- 1010; nextstep,
        SCo=1010 --> (F[2] --> C_count <-- 63, E<2> <--1,
                        (F[1] --> R[4] <-- T_a1[A_M1],
                          F[1] --> R[4] <-- T_a2[A_M2]),
                        SCo <-- 1001; nextstep,
                ¬F[2] --> C_count <-- C_R<5:0>, nop,
                          SCo <-- 0111; nextstep,)
        SCo=1001 --> C_count <-- 0, R[1] <-- R[1] + R[4],
                SCo <-- 1101; nextstep,
        SCo=0111 --> C_count <-- 0, R[1] <-- 0,SCo <-- 0111; nextstep,
        SCo=0110 --> C_count <-- 63, nop,SCo <-- 0101; nextstep,
        SCo=0101 --> C_count <-- 12, nop,SCo <-- 0100; nextstep,
        SCo=0100 --> (F[2] --> C_count <-- 63, E<3> <-- 1,
                          R[4] <-- T_a1[A_M1],
                          SCo <-- 0011; nextstep,
                    ¬F[2] --> C_count <-- 0,
                          R[1] <-- R[1] + R[3],
                          SCo <-- 0000; execute-instruction, ),
        SCo=0011 --> C_count <-- 0, R[1] <-- R[1] + R[4],
                SCo <-- 0110; nextstep,))
¬F[3] --> next (:= (
        SCo=1111 --> C_count <-- 12, R[1] <-- 111..▯R[1]<63:52>,
                R[3] <-- 111...▯R[3]<63:52>,
                SCo <-- 1110; next,
        SCo=1110 --> C_count <-- 52, R[1] <-- 111...▯R[1]<63:12>,
                R[3] <-- 111...▯R[3]<63:12>,
                C_R<5:0> <-- R[1]<5:0>,
                SCo <-- 1101; next,
        SCo=1101 --> C_count <-- 63, nop, SCo <-- 1100; next,
```

```
            SCo=1100 --> C_count <-- 52, nop, SCo <-- 1011; next,
            SCo=1011 --> C_count <-- 12, R[3]<63:0> <-- 0 ▯ R[3]<63:1>,
                         SCo <-- 1010; next,
            SCo=1010 --> (F[2] --> C_count <-- 63, nop,
                                   SCo <-- 1001; next,
                          ¬F[2] --> C_count <-- C_R<5:0>,
                                    R[1] <-- 0 ▯ R[1]<63:1>,
                                    SCo <-- 1000, next,)
            SCo=1001 --> C_count <-- 0, (
                          F[1] --> R[1] <-- R[1] + R[3],
                          F[1] --> R[1] <-- R[1] - R[3],),
                         SCo <-- 1101; next,
            SCo=1000 --> C_count <-- 12,
                         R[1],R[3] <--Neg( R[1]<51:0>) ▯ R[1]<63:52>,
                         SCo <-- 0111; next,
            SCo=0111 --> C_count <-- 52, R[1],R[3] <-- 000..▯R[1]<63:12>,
                         SCo <-- 0101; next,
            SCo=0101 --> C_count <-- 63, E<1> <-- 1,
                         R[2]<0:63> <-- R[1]<63:0>,
                         SCo <-- 0100; next,
            SCo=0100 --> C_count <-- 52,
                         R[2]<63:0> <-- R[2]<11:0>▯R[2]<63:12>,
                         SCo <-- 0011; next,
            SCo=0011 --> C_count <-- 12,
                         R[2] <-- R[2]<0>▯ R[2]<63:1>, F[2] <-- R[2]<0>,
                         R[3] <-- 0 ▯R[3]<63:1>, E<3> <-- 1,
                         A_MI<1:26> <-- 0  A_MI<1:25>,
                         SCo <-- 0010; next,
            SCo=0010 --> (F[2] --> C_count <-- 63, nop,
                                   SCo <-- 0001; next,
                          ¬F[2] --> C_count <-- 0, nop,
                                    SCo <-- 0000; execute-instruction,)
            SCo=0001 --> C_count <-- 0, R[1],R[3] <-- R[1] + R[3],
                         SCo <-- 0101; next,)))
        IN (:=op=24) --> C_count <-- 0, nop,
        OUT (:=op=25) --> C_count <-- 0, nop,);
        execute-instruction, )

InputA := (Id<0> --> InputA <-- R[0]<0>,
           Id<1> --> InputA <-- R[1]<0>,
```

```
          Id<2> --> InputA <-- R[2]<0>,
          ¬(Id<0>∨Id<1>∨Id<2>) --> InputA <-- 0 )

InputB := (Id<3> --> InputB <-- R[3]<0>,
           Id<4> --> InputB <-- R[4]<0>,
           ¬(Id<3>∨Id<4>) --> InputB <-- 0 )

shift := (Id<0> --> R[0]<63:0> <-- R[0]<0>▯R[0]<63:1>,
          Id<1>∨Id<1> --> (Od<1> --> R[1]<63:0> <-- Output▯R[1]<63:1>,
                           ¬Od<1> --> R[1]<63:0> <-- R[1]<0>▯R[1]<63:1>),
          Id<2> --> R[2]<63:0> <-- R[2]<0>▯R[2]<63:1>,
          Id<3>∨Id<0> --> (Od<0> --> R[3]<63:0> <-- Output▯R[3]<63:1>,
                           ¬Od<0> --> R[3]<63:0> <-- R[3]<0>▯R[3]<63:1>),
          Id<4> --> R[4]<52:0> <-- R[4]<52>▯R[4]<52:1> )

repClear := (shift; ( ¬S --> Output <-- 0; repClear))

repMA := (shift; ( ¬S --> Output <-- InputA; repMA))

repMB := (shift; ( ¬S --> Output <-- InputB; repMB))

repAND := (shift; ( ¬S --> Output <-- InputA ∧ InputB; repAND))

repOR := (shift; ( ¬S --> Output <-- InputA ∨ InputB; repOR))

repEXOR := (shift; ( ¬S --> Output <-- InputA.EXOR.InputB; repEXOR))

repMult := (shift; ( ¬S --> Cout <-- Carry(InputA+InputB+Cin),
                            Output <-- Sum(InputA+InputB+Cin); repMult))

repDivA := (shift; ( ¬S --> Cout <-- Borrow(InputA-InputB-Cin),
                            Output <-- Diff(InputA-InputB-Cin); repDivA))

repDivB := (shift; ( ¬S --> Cout <-- Borrow(InputB-InputA-Cin),
                            Output <-- Diff(InputB-InputA-Cin); repDivB))

repIncA := (shift; ( ¬S --> Cout <-- InputA ∧ Cin,
                            Output <-- InputA ⊕ Cin; repIncA))
```

```
repIncB := (shift; (¬S --> Cout <-- InputB ∧ Cin,
                           Output <-- InputB ⊕ Cin; repIncB))

repDecA := (shift; (¬S --> Cout <-- ¬InputA ∧ Cin,
                           Output <-- InputA ⊕ Cin; repDecA))

repDecB := (shift; (¬S --> Cout <-- ¬InputB ∧ Cin,
                           Output <-- InputB ⊕ Cin; repDecB))

repSet1 := (shift; (¬S --> Output <-- Cin, Cout <-- 0; repSet1))

repNegA := (shift; (¬S --> Cout <-- ¬InputA ∧ Cin,
                           Output <-- ¬InputA.EXOR.Cin; repNegA))

repNegB := (shift; (¬S --> Cout <-- ¬InputB ∧ Cin,
                           Output <-- ¬InputB.EXOR.Cin; repNegB))

repSet := (shift; (¬S --> Output <-- 1; repSet1))

repNOTA := (shift; (¬S --> Output <-- ¬InputA; repNOTA))

repNOTB := (shift; (¬S --> Output <-- ¬InputB; repNOTB))
```

APPENDIX 2

Microcode of PLA controller

******I N P U T S******					***********O U T P U T S*****************								
Op	Si	So	Fi	SCi	SCo	CL	Fo	R	Ci	P	K	I/O	E
00001	XXXXX	XXX	XXX	XXXX	0000	100	000	0000	00	0000	0000	00000000	000
00010	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	0011	0000	00000000	000
00011	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	0101	0000	00000000	000
00100	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	0001	0000	00000000	000
00101	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	0111	0000	00000000	000
00110	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	0110	0000	00000000	000
00111	XXXXX	XXX	XXX	XXXX	0000	100	001	0110	00	0110	1000	00000000	000
01000	XXXXX	XXX	XXX	XXXX	0000	100	001	1001	00	1001	0010	00000000	000
01001	XXXXX	XXX	XXX	XXXX	0000	100	001	1001	00	1001	0100	00000000	000
01010	XXXXX	XXX	XXX	XXX0	1000	110	000	0011	00	0011	0000	00000000	000
01010	XXXXX	XXX	XXX	XXX1	0000	010	001	0110	01	0011	1100	00000000	000
01011	XXXXX	XXX	XXX	XXX0	1000	110	000	0011	00	0101	0000	00000000	000
01011	XXXXX	XXX	XXX	XXX1	0000	010	001	0110	01	0101	1010	00000000	000
01100	XXXXX	XXX	XXX	XXX0	1000	110	000	0011	00	0011	0000	00000000	000
01100	XXXXX	XXX	XXX	XXX1	0000	010	001	1001	01	1100	0011	00000000	000
01101	XXXXX	XXX	XXX	XXX0	1000	110	000	0011	00	0101	0000	00000000	000
01101	XXXXX	XXX	XXX	XXX1	0000	010	001	1001	01	1010	0101	00000000	000
01110	XXXXX	XXX	XXX	XXX0	1000	110	000	0000	00	0000	0000	00000000	000
01110	XXXXX	XXX	XXX	XXX1	0000	010	000	0101	01	0000	1111	00000000	000
10000	XXXXX	XXX	XXX	XXXX	0000	100	000	0110	01	1100	0011	00000000	000
10001	XXXXX	XXX	XXX	XXXX	0000	100	000	0110	01	1010	0101	00000000	000
10010	XXXXX	XXX	XXX	XXXX	0000	100	000	1111	00	0000	0000	00000000	000
10011	XXXXX	XXX	XXX	XXXX	0000	100	000	0101	10	0000	0000	00000000	000
10100	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	1100	0000	00000000	000
10101	XXXXX	XXX	XXX	XXXX	0000	100	000	0011	00	1010	0000	00000000	000
XXXXX	1XXXX	XXX	XXX	0000	0000	000	000	0000	00	0000	0000	00000000	100
XXXXX	X1XXX	XXX	XXX	0000	0000	000	000	0000	00	0000	0000	00000001	000
XXXXX	XX1XX	XXX	XXX	0000	0000	000	000	0000	00	0000	0000	00010000	000
XXXXX	XXX1X	XXX	XXX	0000	0000	000	000	0000	00	0000	0000	00001000	000
XXXXX	XXXX1	XXX	XXX	0000	0000	000	000	0000	00	0000	0000	01000000	000
XXXXX	XXXXX	1XX	XXX	0000	0000	000	000	0000	00	0000	0000	00000010	000
XXXXX	XXXXX	X1X	XXX	0000	0000	000	000	0000	00	0000	0000	00100000	000
XXXXX	XXXXX	XX1	XXX	XXXX	0000	000	000	0000	00	0000	0000	10000000	000
0XXXX	1XXXX	XXX	XXX	XXX1	0000	000	000	0000	00	0000	0000	00000000	100
0XXXX	X1XXX	XXX	XXX	XXX1	0000	000	000	0000	00	0000	0000	00000001	000
0XXXX	XX1XX	XXX	XXX	XXX1	0000	000	000	0000	00	0000	0000	00010000	000
0XXXX	XXX1X	XXX	XXX	XXX1	0000	000	000	0000	00	0000	0000	00001000	000
0XXXX	XXXX1	XXX	XXX	XXX1	0000	000	000	0000	00	0000	0000	01000000	000
0XXXX	XXXXX	1XX	XXX	XXX1	0000	000	000	0000	00	0000	0000	00000010	000
0XXXX	XXXXX	X1X	XXX	XXX1	0000	000	000	0000	00	0000	0000	00100000	000
1011X	XXXXX	XXX	XXX	0000	1111	100	100	1001	00	1001	0010	00100000	000
1011X	XXXXX	XXX	XX1	1111	0111	110	000	0011	00	0011	0000	00110000	000
1011X	XXXXX	XXX	XX1	1110	1011	011	001	1111	00	0000	1100	00110000	000
1011X	XXXXX	XXX	XX1	1101	0011	000	000	0000	00	0000	0000	00000100	000

```
1011X XXXXX XXX XX1 1100  1101 010 001 0000 00 0000 1100 00001000 000
1011X XXXXX XXX XX1 1011  0101 110 010 0110 01 0011 1100 00001000 001
1011X XXXXX XXX X11 1010  1001 000 000 0000 00 0000 0000 00000000 010
1011X XXXXX XXX XX1 1001  1011 100 000 0110 00 0110 1000 00110000 100
1011X XXXXX XXX X01 1010  1110 100 000 0000 00 0000 0000 00100000 000
1011X XXXXX XXX X01 0111  1110 100 000 0000 00 0000 0000 00000000 000
1011X XXXXX XXX X11 011X  1010 000 000 0000 00 0000 0000 00000000 010
1011X XXXXX XXX XX1 0101  0110 100 000 0110 00 0110 1000 00110000 100
10110 XXXXX XXX X01 0110  1010 110 000 0011 00 0011 0000 00110000 000
10110 XXXXX XXX XX1 0101  0010 010 001 0110 01 0011 1100 00110000 000
10110 XXXXX XXX XX1 0100  0000 100 000 0110 00 0110 1000 00010011 000
1011X XXXXX XXX XX0 1111  0111 100 000 1111 00 0000 0000 00110011 000
1011X XXXXX XXX XX0 1110  1011 000 000 0000 00 0000 0000 00000000 000
1011X XXXXX XXX XX0 1101  0011 010 000 0000 00 0000 0000 00000000 000
1011X XXXXX XXX XX0 1100  1101 110 000 0000 00 0000 0000 00000010 001
1011X XXXXX XXX XX0 1011  0101 000 000 0000 00 0000 0000 00000000 010
1011X XXXXX XXX 010 1010  0111 100 000 0110 00 0110 1000 00110011 000
1011X XXXXX XXX 110 1010  0111 100 000 1001 00 1001 0010 00110011 000
1011X XXXXX XXX X00 1010  1001 001 000 0000 00 0000 0000 00100000 000
10110 XXXXX XXX XX0 1001  0001 110 000 0011 00 0011 0000 00110011 000
10110 XXXXX XXX XX0 1000  1110 010 001 0110 01 0011 1100 00110011 000
10110 XXXXX XXX XX0 0111  0110 000 000 0000 00 0000 0000 00110011 000
1011X XXXXX XXX XX0 0110  1010 000 000 0000 00 0000 0000 00000100 000
1011X XXXXX XXX XX0 0101  0010 010 001 0000 00 0000 1100 00001000 000
1011X XXXXX XXX XX0 0100  1100 110 011 0110 01 0011 1100 00001000 001
1011X XXXXX XXX X10 0011  0100 000 000 0000 00 0000 0000 00000000 010
1011X XXXXX XXX XX0 0010  0110 100 000 0110 00 0110 1000 00110011 000
1011X XXXXX XXX X00 0011  0000 000 000 0000 00 0000 0000 00000000 000
10111 XXXXX XXX X01 0110  0000 100 000 0110 00 0110 1000 00010011 000
10111 XXXXX XXX XX0 1001  0001 100 000 0110 01 1100 0011 00110000 000
10111 XXXXX XXX XX0 1000  1110 110 000 0011 00 0011 0000 00110011 000
10111 XXXXX XXX XX0 0111  0110 010 001 0110 01 0011 1100 01010011 000
1100X XXXXX XXX XXX XXXX  0000 000 000 0000 00 0000 0000 00000000 000
```

Discription of Input/Output of PLA

Input:

Op<0:4>	!5-bit op code;
Si<0:4>	!5-bit input selection lines, which select one, two or none inputs from five registers;
So<0:2>	!3-bit output selection lines, So<0,1> choose either or both of R[1] and R[3] as the output of ALU, So<2> is used only at "OUT" instruction;
Fi<1:3>	!3-bit Flag input lines from the content of F<1:3>;
SCi<1:4>	!4-bit Sequence Control lines, which is the number of micro-steps ahead for the current instruction; the only state variables in this PLA;

Output:

SCo<4:1>	!4-bit Sequence Control lines, which is one less than the value of SCi. These four lines feedback to SCi. When SCo is counted down to zero, the current instruction is done and next instruction can be read from input pads.
CL<1:3>	!3-bit Counter Loading output lines, which choose the initial value of counter, so that the cycle time of present micro-step can be set.
Fo<1:3>	!3-bit Flag setting lines, each line corresponds to one of Flags and selects one of two input sources.
R<4:1>	!4-bit Run control lines of ALU;
Ci<2:1>	!2-bit Carry input selection lines;

P<4:1> !4-bit Propogate control lines of ALU;

K<4:1> !4-bit Kill control lines of ALU;

I/O<7:0> !8-bit Input or Output selection lines, which correspond to the inputs Si and So. To make the physical routing simple, the input and output lines mix together in random order.

E<3:1> !3-bit Enable lines, which control one of three operations: 1) parallel loading from R[2] to R[3], 2)table look-up and 3)table address shifting.

APPENDIX 3

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                          2.0000
Input data B:                          4.0000

The log value of A is:                 1.0000000000000E+00
The log value of B is:                 2.0000000000000E+00

The difference of logA and LogB:      -1.0000000000000E+00

The value of log (A+B) is:             2.5849625007212E+00
The number of iterations is:           30
Total clock cycles is:                 3313

The answer by direct computation:      2.5849625007212E+00
```

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                          2.0000
Input data B:                          6.0000

The log value of A is:                 1.0000000000000E+00
The log value of B is:                 2.5849625007212E+00

The difference of logA and LogB:      -1.5849625007212E+00

The value of log (A+B) is:             3.0000000000000E+00
The number of iterations is:           14
Total clock cycles is:                 1630

The answer by direct computation:      3.0000000000000E+00
```

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

Input data A:	4.0000
Input data B:	12.0000
The log value of A is:	2.0000000000000E+00
The log value of B is:	3.5849625007212E+00
The difference of logA and LogB:	-1.5849625007212E+00
The value of log (A+B) is:	4.0000000000000E+00
The number of iterations is:	13
Total clock cycles is:	1501
The answer by direct computation:	4.0000000000000E+00

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

Input data A:	40.0000
Input data B:	216.0000
The log value of A is:	5.3219280948874E+00
The log value of B is:	7.7548875021635E+00
The difference of logA and LogB:	-2.4329594072761E+00
The value of log (A+B) is:	8.0000000000000E+00
The number of iterations is:	40
Total clock cycles is:	4442
The answer by direct computation:	8.0000000000000E+00

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                        92.0000
Input data B:                      8100.0000

The log value of A is:               6.5235619560570E+00
The log value of B is:               1.2983706192659E+01

The difference of logA and LogB:    -6.4601442366023E+00

The value of log (A+B) is:           1.3000000000000E+01
The number of iterations is:         46
Total clock cycles is:             5032

The answer by direct computation:    1.3000000000000E+01
```

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                      4002.0000
Input data B:                      4190.0000

The log value of A is:               1.1966505451906E+01
The log value of B is:               1.2032734528587E+01

The difference of logA and LogB:    -6.6229076680976E-02

The value of log (A+B) is:           1.3000000000000E+01
The number of iterations is:         41
Total clock cycles is:             4545

The answer by direct computation:    1.3000000000000E+01
```

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                        1024.0000
Input data B:                        7166.0000

The log value of A is:                 1.0000000000000E+01
The log value of B is:                 1.2806952328211E+01

The difference of logA and LogB:      -2.8069523282107E+00

The value of log (A+B) is:             1.2999647736528E+01
The number of iterations is:           51
Total clock cycles is:                5591

The answer by direct computation:      1.2999647736528E+01
```

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                          6.6670
Input data B:                          9.3330

The log value of A is:                 2.7370377271149E+00
The log value of B is:                 3.2223408955935E+00

The difference of logA and LogB:      -4.8530316847851E-01

The value of log (A+B) is:             4.0000000000000E+00
The number of iterations is:           46
Total clock cycles is:                5040

The answer by direct computation:      4.0000000000000E+00
```

SIMULATION OF LOGARITHMIC ADDITION ALGORITHM

```
Input data A:                          90.4556
Input data B:                         112.3547

The log value of A is:                  6.4991379160549E+00
The log value of B is:                  6.8119166660938E+00

The difference of logA and LogB:       -3.1277875003882E-01

The value of log (A+B) is:              7.6639871132283E+00
The number of iterations is:           40
Total clock cycles is:                4431

The answer by direct computation:      7.6639871132283E+00
```